NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
NWU-CS-02-12
September 26, 2002

An Implementation of Diffusion in the Linux Kernel

Brian Cornell **Jack Lange** **Peter Dinda**

Abstract

Packet headers and trailers exhibit considerable coding redundancy from both a theoretical and a practical standpoint. Diffusion exploits this redundancy to create an additional communication channel between hosts as a by-product of normal packet transfers. This channel is zero cost: the number and size of packets transferred do not change. Information is piggybacked on existing packets by overwriting unused fields in their headers and trailers. Dissemination of dynamic resource information is a natural use of such a channel.

This paper describes the interface, implementation, and performance of Diffusion on the Linux operating system. In addition, it describes two tools, SpyTalk and LoadBanner, which have been built on top of this implementation. It is also a user manual for those who wish to try Diffusion.

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, and EIA-0130869, including a Research Experience For Undergraduates (REU) Supplement. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF).

Keywords: information dissemination, resource monitoring, networks

An Implementation of Diffusion in the Linux Kernel

Technical Report NWU-CS-02-12

Brian Cornell Jack Lange Peter Dinda
cornell, jrl829, pdinda@cs.northwestern.edu
Department of Computer Science
Northwestern University

September 26, 2002

Abstract

Packet headers and trailers exhibit considerable coding redundancy from both a theoretical and a practical standpoint. Diffusion exploits this redundancy to create an additional communication channel between hosts as a by-product of normal packet transfers. This channel is zero cost: the number and size of packets transfered does not change. Information is piggybacked on existing packets by overwriting unused fields in their headers and trailers. Dissemination of dynamic resource information is a natural use of such a channel.

This paper describes the interface, implementation, and performance of Diffusion on the Linux operating system. In addition, it describes two tools, SpyTalk and LoadBanner which have been built on top of this implementation. It is also a user manual for those who wish to try Diffusion.

keywords: information dissemination, resource monitoring, networks

1 Introduction

Resource monitoring and prediction systems such as Remos [3], NWS [5], RPS [2], and GMA-based systems [4] collect information about resource behavior on the behalf of applications, users, and other middleware. An important challenge in these systems is disseminating the information they collect to these interested parties as efficiently as possible, making minimal use of network resources.

In an earlier paper [1], we considered the extremes of this challenge: could we communicate resource information with *no* use of network resources, exploiting existing packet transfers? The paper demonstrated that the prospects for doing so are quite good. We found that existing packets exhibit considerable redundancy from an information-theoretic perspective. We further identified a number of mechanisms that could be practically applied to exploit this redundancy while still keeping packets compatible with existing networks. We collectively refer to the use of these mechanisms to disseminate information as *Diffusion*.

Our mechanisms are essentially based on overwriting unused fields within packet headers and trailers with the data that we want to convey. For example, if an outgoing TCP packet does not have its URG flag set, its 2 byte urgent pointer field is unused. We would overwrite it with two bytes of data that we want to

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, and EIA-0130869, including a Research Experience For Undergraduates (REU) Supplement. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF).

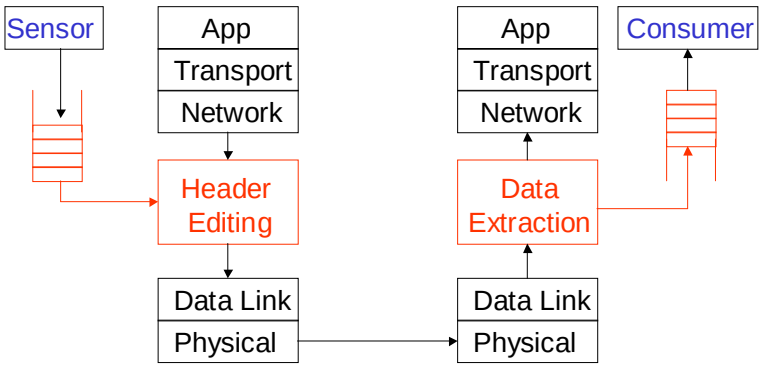

Figure 1: A high-level view of Diffusion and where it sits in the network stack.

send to the host for which the packet is destined. Correspondingly, an incoming packet without the URG flag set could be stripped of its urgent pointer if we are listening for messages from the source host. Figure 1 illustrates where our scheme fits in the network stack. Including the TCP urgent pointer, we identified 10 different mechanisms involving TCP and IP headers, and Ethernet padding.

Our earlier work relied on a proof-of-concept test using a user-level network stack. Since the prospects seemed quite bright, we decided to implement Diffusion within a commonly used operating system. We chose Linux because of its ubiquity in many of the domains in which resource monitoring systems are used, our general familiarity with it, the ready availability of source code, and the convenient `/proc` file system interface it provides.

The Linux implementation of Diffusion has essentially four parts. First, there are small modifications to the existing kernel code that let us hook into incoming and outgoing packet processing. Second, there are two kernel modules, one for sending data, one for receiving data. The source code for these modules is separate from the main kernel sources, and the modules themselves can be inserted into the running kernel when they are needed. The modules export a simple, but powerful send and receive interface via the `/proc` file system. The interface allows applications to define and install send-side and receive-side filters that are similar to routing table entries. They can then queue Diffusion data to be sent when packets matching those rules are sent. On the receive side, incoming packets which match the rules have their attached Diffusion data queued for delivery to applications via files in `/proc`. Applications can read and write the `/proc` interface directly, but we also provide a C interface, which forms the third part of the implementation. Finally, our implementation includes two example applications: *SpyTalk*, a version of the Unix talk program that communicates surreptitiously using Diffusion; and *LoadBanner*, which uses Diffusion to keep clients apprised of the load situation of hosts on the LAN.

Linux Diffusion does not implement all the mechanisms described in the earlier paper. It implements: using the IP identifier field for packets with the don't fragment flag set, using the reserved IP flag, using the TCP urgent pointer when the urgent flag is not set, using the TCP reserved flags, and using Ethernet minimum length padding. In addition, it provides a convenient interface for adding information to broadcast ARP request packets.

In the following, we first describe how to install Diffusion on a Linux machine. Next, we give an example of using the system. This is followed by a detailed discussion of its components, starting with the kernel modules, then covering the /proc interface, and finally the C interface. We follow this with a description of the SpyTalk and LoadBanner applications. Finally, we present measurements of the overhead involved in Diffusion, finding that it has a negligible effect on performance provided the number of filters installed by applications is moderate.

2 Installing Diffusion

This section describes what is needed to run Diffusion, where to get it, and how to build and install it.

2.1 Requirements

Diffusion requires a relatively current kernel version with source. It has been tested on kernel versions 2.4.18 and 2.4.19. To get a new version of the Linux kernel, visit http://kernel.org. Diffusion also requires that the kernel is built with IPv4 networking support, and module support. You will also need a recent version of gcc, gnu make, and Perl 5.

Our specific environment is:

```
gcc version 2.96 20000731 (Red Hat Linux 7.1 2.96-98)
GNU Make version 3.79.1, by Richard Stallman and Roland McGrath.
  Built for i386-redhat-linux-gnu
Red Hat Linux release 7.2 (Enigma)
Linux parakeet 2.4.19 #1 SMP Thu Aug 15 17:31:49 CDT 2002 i686 unknown
This is perl, v5.6.0 built for i386-linux
patch 2.5.4
```

2.2 Getting and unpacking Diffusion

Diffusion is provided with no warranties or guarantees of any kind and we can not provide support for it at this time. Please understand that Diffusion is a piece of *research code* that involves *modifying your kernel* and the normal data path of your network stack. Bugs or unfortunate interactions with other software could lead to *kernel panics, crashes, and data loss.* Furthermore, Diffusion will inject *modified packets* into your network. While we believe that these packets will cause no problems, we can not guarantee this. Running Diffusion *could cause network problems.* If your computer explodes and your network catches on fire, please don't come running to us.

To get a copy of Diffusion, please visit http://www.cs.northwestern.edu/∼plab/Diffusion. That page will describe how to get the Diffusion tarball, `diffusion-version.tgz`. Run

```
tar xvfz diffusion-version.tgz
```

to extract the tarball into the `diffusion-version` directory.

2.3 Compilation and installation

There are four simple steps to installing the Diffusion kernel modifications and modules. You will need to be root to execute all but the third step.

First, you must install the provided kernel patch. This patch adds packet intercepting abilities needed by Diffusion to the kernel. To install the patch, change into the directory where you've extracted Diffusion and type

```
make patch
```

This will apply the patch to the currently running kernel version, assuming that your kernel source is in the directory `/usr/src/linux-`*kernelversion*.

Second, you must make a new kernel. Your kernel should come with documentation concerning the configuration and compilation of the kernel. You must be running this new kernel version before you continue installing Diffusion.

Third, you must compile the Diffusion modules. To do this, change into the directory where Diffusion was extracted and type

```
make
```

Finally, you should install the modules into the system's module path. To do this, type

```
make install
```

This will copy the modules into `/lib/modules/`*kernelversion*, where `insmod` will look for them.

2.4 Loading the kernel modules into a running kernel

Loading the modules into a running kernel is easy after you have completed the above installation procedure. Simply enter the following commands as root:

```
insmod ipdsend
insmod ipdrecv
```

Once loaded, the modules can also be removed easily. Again, type the following commands as root:

```
rmmod ipdsend
rmmod ipdrecv
```

To determine whether the modules are currently loaded into the kernel, issue the following command (you need not be root this time, though you will have to specify the path (`/sbin`) if you're not):

```
lsmod
```

3 Examples of using Diffusion

Before we go into detail about the inner workings of Diffusion, let's first take a look at how the system works using several examples. As an introduction and a test of the system, we'll perform a couple of tasks using SpyTalk. If you don't have SpyTalk installed, change into the SpyTalk directory of the distribution and type `make` followed by `make install`. For more details about SpyTalk, see Section 7.1. Following the SpyTalk examples, we'll give examples of using the `/proc` interface directly.

3.1 SpyTalk Examples

Example 1: Text Communication In the first example, we will open a text communication channel between two SpyTalk sessions running on the same host. No data is actually communicated in Diffusion unless there is existing network traffic on which to piggyback. In this example and the ones that follow, we will create fake network traffic using repeated ICMP ping requests and responses. You will need three terminals, two of which need root access, to complete this example.

To begin, on one of the terminals (this one does not need root access), type

```
ping -i .05 <ipaddr>
```

where `<ipaddr>` is the IP address of your own computer. This simply produces the ping request packets and corresponding response packets on which we'll piggyback. You can leave this terminal alone now, we won't be needing it.

Next, we'll start listening for messages. To do this, type

```
SpyTalk -l <ipaddr>
```

on one of the terminals with root access. This terminal is where we should see our messages that we're about to send.

Now on the third terminal (also with root access), type

```
SpyTalk -t <ipaddr>
```

This will complete the communication channel. Type anything you want on this terminal now. You should see it slowly transfer to the terminal running the listener. If you don't, check your installation of Diffusion and SpyTalk.

You may notice that the receiving end misses a character every once in a while. This is normal. The Diffusion communication channel is unreliable: data can be lost, corrupted, or arrive out of order. SpyTalk takes measures to increase the reliability of the channel Diffusion creates, but even here there are no guarantees and some of these measures only work well in interactive mode.

When you are satisfied that Diffusion is working, close both the sending and receiving sessions of SpyTalk by pressing ctrl-D in each. If you wish to try more examples, leave the ping running, otherwise close it with ctrl-C.

You can extend this example (and the ones that follow) in obvious ways to communicate between two different hosts equipped with Diffusion. The sending SpyTalk is simply given the IP address of the receiving host, while the receiving SpyTalk is given the address of the sending host. In interactive mode, SpyTalk combines two channels for two-way communication, giving the user an interface similar to the Unix talk program.

Example 2: Sending a File This second example is similar to the first. You will need three terminals again. Start the ping process on the first terminal as before. On a second terminal, run the SpyTalk listener again just as above. On the third terminal we will do something slightly different.

First, create a small file with some text in it. There are many ways to do this: using your favorite editor (vi, emacs, etc.), using echo redirected into a file, etc. Once you have created this file, type

```
SpyTalk -tf <filename> <ipaddr>
```

where `<filename>` is the text file you created, and `<ipaddr>` is your IP address. Now switch to the terminal running the receiver, and you should see the file you wrote appear byte by byte. Again, there may be gaps in the file every now and then. Again, this is normal given the unreliable communication channel.

When the file finishes sending, the sending side of SpyTalk will automatically close itself. When it does so, go ahead and close the receiving end with ctrl-D. Leave the ping running if you plan to run any more examples. Run `SpyTalk --help` for other SpyTalk options.

3.2 /proc Example

SpyTalk interacts with the kernel component of Diffusion via the `/proc` interface. In the following, we'll show how to use that interface directly. Before you start, you need to manually insert the modules. You should also start a ping process or other data transfers between the hosts you plan to use.

Example 3: Queuing and Receiving In this example, we will queue data for sending using the `/proc` interface, and create a listener to receive it. If it is not already running, start ping as described in Example 1. You will need one terminal with root access to complete this example.

With ping running, first we need to create a listener to receive anything we might send. We'll create two such listeners, one for each protocol. To do this, type the following two lines:

```
echo "add <ipaddr>/32 IP_DFRAG ip N H" > /proc/ipd_ctrl
echo "add <ipaddr>/32 TCP_URGENT tcp N H" > /proc/ipd_ctrl
```

To ensure that the listeners have been created correctly, list the directory `/proc/ipd/`. You should see two files: `ip` and `tcp`. The first command tells Diffusion to listen for packets coming from `<ipaddr>` (the /32 is a subnet mask). When such an IP packet arrives and its don't fragment bit is set, it will strip out the packet identifier and reserved bit and treat them as data. The data will be queued for the application to read within the file `/proc/ipd/ip`. The second command is similar, but it extracts data from the TCP header and queues it to the file `/proc/ipd/tcp`. Section 5 describes the `/proc` receive interface in much more detail.

Now let's queue some data to be sent. We'll use Diffusion to send "Ok" using both the TCP and IP layer mechanisms. To do this, type

```
echo -e "\3\0\0\0\0\0\377\0Ok\31" > /proc/ipdsend
```

This will put "Ok" in the queue to be repeatedly sent to any target on the IP and TCP protocols. In essence, this command is similar to the receive side commands, except represented in binary. To check that it worked (and display the entry in text), type

```
cat /proc/ipdsend
```

to see what is in the send queue. For details on how to interpret this data see the section on the `/proc` interface below. To see if the data is being received, type

```
cat /proc/ipd/*
```

or

```
head /proc/ipd/*
```

This will show the contents of the receive queues. You should see it filling up with entries.

Notice that the send interface to Diffusion is a binary interface, while the receive interface is text-based. This is an artifact of the implementation. We intend at some point to provide both binary and text interfaces for both send and receive.

Try some of the following commands and see what effects they have on the Diffusion system.

```
echo -e "\1\0\0\0\0\0\377\0IP\20" > /proc/ipdsend
echo -e "\2\0\0\0\0\0\377\0TC\12" > /proc/ipdsend
echo -e "\3\0\0\0\0\0\144\0:.\31" > /proc/ipdsend
echo -e "\203" > /proc/ipdsend
echo -e "\3\0\0\0\0\0\377\50!!\0" > /proc/ipdsend
echo " " > /proc/ipd/ip
echo "anything" > /proc/ipd/tcp
echo "del ip" > /proc/ipd_ctrl
echo "del tcp" > /proc/ipd_ctrl
```

For more details on how to use the send and receive sides of the `/proc` interface, see Section 5.

4 Kernel implementation

This section describes our modifications to the Linux kernel and the kernel modules that implement Diffusion.

4.1 Kernel modifications

The modifications to the kernel consist of two functions, exported for use by the Diffusion modules. The functions are `register_send_packet_interceptor` and `register_recv_packet_interceptor`. They each take a function as an argument. Whenever the kernel is about to send a packet, or has just received a packet, it will call the corresponding registered "interceptor" function. The interceptor functions are passed a pointer to the `sk_buff` struct pertaining to the packet. Only one send and one receive interceptor function may be registered at a time. Registering a new function overwrites the old one. Registering NULL causes the no interceptors to be run.

These modifications are implemented through the addition of code to `net/core/dev.c`, an integral part of the kernel networking subsystem. In addition, the makefile containing dev.c is changed to allow the functions to be exported. Our kernel patch installs these modifications.

4.2 Send module

The send module, `ipdsend`, uses `register_send_packet_interceptor` to register itself to intercept all outgoing packets. It modifies each outgoing packet according to send requests that have been queued to it from applications.

When loaded, the send module creates a `/proc` file, `/proc/ipdsend`, to use as an interface with higher level programs, and registers an internal handler function as the interceptor. It also creates two empty linked list structures to serve as queues, one for IP and TCP data, and the other for generic Ethernet data. In addition, it creates a 60-byte static buffer for ARP data which is initially zeroed.

When a packet is sent out of the system while the send module is running, it goes through a series of checks in the handler. First, the handler determines whether the `sk_buf` structure it is passed contains an IP packet, and if so, whether it contains a TCP segment. It also detects whether the packet is an ARP request packet. The packet is then passed to handlers for each of the protocols that match. After the appropriate handlers (if any) are done with the packet, it is sent to the generic Ethernet handler which may pad its length using queued Ethernet data. These steps are done with interrupts disabled and while under a spin lock.

IP Header editing When the IP packet handler receives a packet, it first checks to see if the Don't Fragment flag is set. If not, it leaves the packet alone. If it is, it then selects the next data item that should be sent given the destination of the packet from the queue. As much as possible of the data is written into the ID field and the reserved flag of the IP header. The handler then uses the `ip_fast_csum` function to fix the checksum in the IP header.

TCP Header editing The TCP handler works similarly. It checks to see if the urgent flag is set, and doesn't touch the packet if it is. If not, it gets the next data item that should be sent given the destination from the queue and writes as much as possible into the urgent pointer field and reserved flags of the TCP header. The handler then uses the functions `csum_tcpudp_magic` and `csum_partial` to fix the TCP and IP checksums.

Selecting the data item to send Both the IP and TCP handlers must select the most appropriate data item to send from among all the data applications have queued. To do this, we search through the corresponding queue (IP or TCP) of data looking for matches to the packet's destination address and protocol. Queued data contains a network address (IP address and netmask), protocol, repeat count, and priority. If there are multiple matches, they are singled out to find the one with (in order of importance) the largest netmask (the most specific to the destination), the highest priority, and the fewest number of times left to repeat. If there are still multiple matching entries, the last one on the queue (the oldest) is selected. After a data item has been selected, its repeat count is decreased by one (unless it is infinite), and it is returned to the handler to be written into the packet. If an entry with a repeat count of zero is ever found in the process of searching, it is removed from the queue. If no matching entries can be found, a default blank entry is returned.

ARP padding Since all ARP requests are sent to the broadcast Ethernet address, only a single data item is ever available to be sent. The ARP handler simply tests the size of the packet to see if it's less than 60 bytes (64 when actually sent). If the packet is small enough, it increases the size of the packet to 60 bytes and fills the empty space with the beginning of the ARP data buffer (however much will fit). In addition, it is possible to queue data on ARP requests in a similar way to IP and TCP by using the generic Ethernet padding handler. Simply set the ARP mechanism to send zero length data and then queue one or more generic Ethernet entries with a mask of 0 and protocol of `0x0806`.

Generic Ethernet packet padding The generic Ethernet handler is similar to the IP and TCP handlers in the method of obtaining data, but similar to ARP in the method of modifying the packet. This handler first determines if the packet is small enough that it needs to be padded (like the ARP handler). If it is, then it searches for the most appropriate data item to send. The search process is identical to that of IP and TCP except that it matches on Ethernet addresses. If there is no matching data, it does not pad the packet.

Limitations In the current implementation, the send module traverses a linked list of outstanding data items searching for the most appropriate. Because of this it is a good idea to try to keep the queue short. A good rule would be to keep the queue under 500 messages in length. Applications built on top of Diffusion should monitor this and control their rate accordingly. Note that this is a limitation of the current implementation. In essence, the search is a longest prefix match similar to that done in an IP router. There are well known techniques for making this a logarithmic or even expected constant time operation.

The send module has an elaborate `/proc` interface for working with the queues. This interface is described in detail below in Section 5.

4.3 Receive module

The receive module uses a function hook in the kernel network stack to intercept and examine all the inbound network packets to the system.

Listeners The receive module operates by hosting a set of user-defined listeners. These listeners are defined to look for certain packets sent from certain network addresses. Each listener can look for network traffic from a single host or multiple hosts in the same network block. Each listener has its own queue associated with it that holds all the messages that are received by the system. These queues are accessible from `/proc` files defined by the user. Each queue has a set length and does not grow. This means that each queue will only hold the last n messages received by the module, while the older messages are discarded.

Interface The module's interface is implemented with a set of proc files that accept input and provide output for the system. A control file is created in the top level of the `/proc` directory, while a subdirectory is created to hold the output files for any listeners defined for the system. Listeners can be created and deleted through the control file, and their output files are taken care of by the system.

The message queues are accessible by reading proc files. Each listener defines a filename (that must be unique) when it is added to the system. This filename is manifested in the directory created by the receive module. Whenever the file is read the current state of the queue is written to the output. The file itself has several configuration flags that can be set when a listener is defined. The module can be configured to clear out the message queue whenever the output file is read, or the queue can be left as is. If the latter is chosen new messages are put onto the end of the queue, while the oldest messages are popped off the other end. The output files also support two formats for outputting the queue. The first format is in human readable form, meaning the values are converted to ASCII before they are written out. The second format is straight binary that goes through no conversion, and is designed to be best suited for client programs making use of the transmission medium. This format is used for the C API that has been written to interface into the system from client programs.

The listener definitions can have duplicate hosts among them. Such that a listener defined for a single host 1.2.3.4 and another listener defined for the subnet 1.2.3.0/24 will each have a copy of any message sent from the host 1.2.3.4, assuming they are both defined for the same protocol. The support of different protocols allows up to three simultaneous communication channels to be active at once. Expanding that number would require another protocol layer to be written on top of the channel.

Supported protocols Currently the module supports 3 protocols as transmission mediums: IP, TCP, and ARP. Any packet from one of these protocols is sent into the system for further processing. The module then cycles through all of its listeners checking the source IP address of the packet against the addresses defined by each listener. The search process is similar to that of the send module.If there is a match the module then checks header flags (for IP and TCP) or byte ranges (for ARP) to determine if the packet does indeed contain a message. If all three of those checks pass, the data is extracted from the header fields and put into the message queues of the appropriate listeners.

Notice that the receiving module does not currently support generic Ethernet padding. Unlike in the sending module, the receiving module would have to parse the packet contents to determine the "real" length of the data. The receiving module does include hooks whereby such a parser could be added.

5 /proc interface

There are two `/proc` interfaces for the Diffusion kernel modules, one for each kernel module. They control all aspects of sending and receiving using Diffusion. The send interface, `/proc/ipdsend`, allows you to queue data to be sent in outgoing packets, clear the buffers of unsent data, and retrieve the status of the buffers. The receive interface, `/proc/ipd_ctrl`, allows you to create and delete listeners. Received messages are also outputted into files located in `/proc/ipd/`.

5.1 Queuing data to be sent on IP and TCP headers

Because of their similarity, the IP and TCP protocols share a queue in the send module. Entries in this queue indicate whether they are IP or TCP packets, or both. Entries may be added to this queue by writing to the file `/proc/ipdsend` in the following 11-byte binary format:

Byte 1: Protocols

A (binary) value of 1 here will queue the data for IP. 2 will queue it for TCP. These values are not exclusive; a value of 3 will queue the data for both IP and TCP.

Bytes 2-5: IP address

The destination IP address or network. Encoded in binary so that "ABCD" would be the IP address 65.66.67.68.

Byte 6: Network mask

The number of bits of the IP address to pay attention to. This must be between 0 and 32. A value of 0 tells the module to ignore the IP address and send this data to anybody. A value of 32 tells it to only send this data to the single IP address specified. Common values include 24 for a class C subnet, 16 for a class B subnet, and 8 for a class A subnet.

Byte 7: Repeat

This is a signed character telling the module how many times to send this data before deleting it from the queue. A value of 0 will not send this data at all, a value between 1 and 127 will send it that number of times, and a value of -1 (255) will send it forever until the buffer is cleared or new infinitely repeating data replaces it. If there is already infinite data queued for this IP address and mask, it will be removed from the queue.

Byte 8: Priority

This is a signed character telling the module what priority to attach to the data. Higher priority data will be sent before lower priority data, even if the lower priority data was there first. The highest priority is 127, and the lowest is -128.

Bytes 9-10: Data

This is the data that will be put in the two-byte area of the packet. For IP packets, this data will be put in the ID field if the packet has the Don't Fragment flag set. For TCP packets, this data will be put in the Urgent Pointer if the packet does not have the urgent flag set.

Byte 11: Flags

This is the data that will be put in the reserved flags of the packet depending on the protocol. For TCP, there are four reserved flags, so the lower four bits of this byte will be put in those flags. If the data is put in the IP header, these four bits will be ignored. For IP, there is one reserved flag, so the next bit of this byte (<Byte 11> AND 0x10) will be put in that flag. This bit will be ignored if the data is put in the TCP header.

Data to be sent is selected based on a series of rules. First, the send module looks for the data with the longest prefix match to the outgoing destination address. In other words, the most specific match wins. If there is a tie, the module chooses the data with the highest priority. If a tie remains, the data that has smallest repeat count is chosen. If there is still a tie, the oldest data is chosen.

5.2 Queuing data to be sent via Ethernet padding

Data to be sent on generic Ethernet packets is queued in a manner similar to that of IP and TCP data. The format for enqueuing Ethernet consists of the following 13+ bytes, which are to be written to `/proc/ipdsend`:

Byte 1: Protocol

A (binary) value of 8 is needed to tell the module that this is Ethernet data.

Bytes 2-7: MAC address

The destination MAC address or network. Encoded in binary so that ”ABCDEF” would be the MAC address 41:42:43:44:45:46.

Byte 8: Network mask

The number of bits of the MAC address to pay attention to. This must be between 0 and 48. A value of 0 tells the module to ignore the MAC address and send this data to anybody. A value of 48 tells it to only send this data to the single MAC address specified. Note that the Ethernet address space is not hierarchical, so most data will probably be queued to specific addresses or to everyone, not ranges. However, it may be useful to queue to a range of multicast addresses.

Byte 9: Repeat

This is a signed character telling the module how many times to send this data before deleting it from the queue. A value of 0 will not send this data at all, a value between 1 and 127 will send it that number of times, and a value of -1 (255) will send it forever until the buffer is cleared or new infinite data replaces it. If there is already infinite data queued for this MAC address and mask, it will be removed from the queue.

Byte 10: Priority

This is a signed character telling the module what priority to attach to the data. Higher priority data will be sent before lower priority data, even if the lower priority data was there first. The highest priority is 127, and the lowest is -128.

Bytes 11-12: Ethernet Protocol

These two bytes identify the Ethernet protocol to pad with the given data. The module will look at the protocol of each packet and only use this data if it matches the protocol defined here.

Byte 13: Data size

This byte identifies how much data is being queued. It may range from 0 to 60, and identifies the size of the data section (bytes 14+). It is not guaranteed that all of the data will be sent, however. The module will only send the first n bytes, where n is the number of bytes that the packet needs to be padded with to be valid.

Bytes 14+: Data

This is the data, copied in binary. There must be exactly as many bytes as the data size says there will be, and it must not exceed 60.

Data will be selected from the queue based on essentially the same rules as for IP and TCP packets.

5.3 Queuing data to be sent via ARP request padding

Since ARP request packets are always broadcast, there is no need to define a destination. Therefore, ARP does not have a queue in the send module, it only has a string. This string is always sent in any ARP packet in the same way that other Ethernet padding data is sent. To set this string write to `/proc/ipdsend/` using the following simple binary format:

Byte 1: Protocol

A (binary) value of 4 is needed to tell the module that this is ARP data.

Byte 2: Data size

This byte identifies how much data is being queued. It may range from 0 to 60, and identifies the size of the data section (bytes 3+). It is not guaranteed that all of the data will be sent, however. The module will only send the first n bytes, where n is the number of bytes that the packet needs to be padded with to be valid.

Bytes 3+: Data

This is the data, copied in binary. There must be exactly as many bytes as the data size says there will be, and it must not exceed 60.

5.4 Clearing send buffers

Send queue entries normally clear automatically with time. As repeat counts reach 0, the entries are removed. The exception is infinitely repeating data. Because of this, there is a method for removing infinite data. When data is queued that is identical to a currently queued infinite entry, but has a different repeat count, the infinite data is removed. It must have the same destination, mask, priority, protocol, and data in order to replace the infinite data.

Because queues can get quite large and full of unwanted data, there is also a method for clearing the entire queue for a protocol. If a single byte is written to the `/proc/ipdsend` file, with the highest bit set, the module clears the queues specified by the lower bits. If the lowest bit is set, all IP and TCP packets that would be sent via IP are deleted. If the second to lowest bit is set, all IP and TCP packets that would be sent via TCP are deleted. If the third lowest bit is set, the ARP message is reset to all zeros. If the fourth lowest bit is set, the Ethernet queue is cleared. The other three bits are as of yet unused.

5.5 Getting send module status

The status of the send module and the contents of its queues can be retrieved by reading the `/proc/ipdsend` file. The output of this file is a human readable list of all the data queued for transmission with the following format:

```
IP/TCP Entries: <IPTCP> ETH Entries: <ETH>
<ipaddr>                <prior> <data>          x<rpt>  <protos>
...
<macaddr>               <prior> <proto> <rpt>x  <len>:<ethdata>
...
<arpdata>
```

Where the bracketed elements have the following meanings:

`<IPTCP>`

The number of IP and TCP entries in the queue, listed in newest to oldest order directly below this line.

`<ETH>`

The number of Ethernet entries in the queue, listed in newest to oldest order after the oldest IP or TCP entry.

`<ipaddr>`

The destination IP address and mask for the data.

`<prior>`
: The priority number of the data.

`<data>`
: The two bytes of data that will be put in the ID or urgent pointer of the packet, followed by the one bit that will be put in the reserved flag of an IP packet and the four bits that will be put in the reserved flags of a TCP packet.

`<rpt>`
: The number of times remaining that this data will be sent or `Inf` for an infinite packet.

`<protos>`
: The letter `I` here means the data will be put in IP packets. The letter `T` means it will be put in TCP packets.

`<macaddr>`
: The destination MAC address with an IP style mask.

`<proto>`
: The Ethernet protocol number to use this data for.

`<len>`
: The length, in bytes, of the data being sent.

`<ethdata>`
: The data to send displayed as a string.

`<arpdata>`
: The data that will be broadcast in all outgoing ARP packets.

5.6 Creating a listener

Listeners are created by issuing a command into the control file `/proc/ipd_ctrl`.

```
add <ip address>/<netmask> <transport> <filename> <clear> <format>
```

`<ip address>`
: The ip address to listen for in dotted notation. Combined with the netmask, this chooses which packets to intercept from the system.

`<netmask>`
: The netmask to apply to the given IP address. Must be an integer value between 0 and 32.

`<transport>`
: The transport mechanism which the listener will look for. Can be one of three values (`ARP`, `IP_DFRAG`, or `TCP_URGENT`).

`<filename>`
: The file which the listener will report its results to. This must not include a path name as it will be automatically created in `/proc/ipd/`.

`<clear>`
Either Y or N. A flag that tells the module whether to clear the queue when the file is read. Y will clear, N will leave the queue intact.

`<format>`
Specifies the output format of the listener. Values can be either `H` or `M`. These correspond to Human and Machine readable output. Human readable is printed out in ASCII, while Machine readable is printed out in straight binary.

5.7 Deleting a listener

```
del <filename>
```

`<filename>`
The name of the output file used by the listener in /proc/ipd/, which is the same as the filename given in the add command.

5.8 Getting the receive module status

```
Listeners: <# of listeners>
IP               PROTO          FILENAME      BLANK_ON_READ   FORMAT
<ip>/<netmask>  <transport>    <filename>    <Y/N>           <H/M>
...
```

`<# of listeners>`
The number of listeners currently active on the system.

`<ip address>`
The ip address that the listener is looking for.

`<netmask>`
The netmask that is applied to the given ip address to match incoming packets.

`<transport>`
The transport mechanism used by the listener. `ARP`, `IP_DFRAG`, or `TCP_URGENT`.

`<filename>`
The filename the listener is using to output the messages it receives. Located in `/proc/ipd/`.

`<Y/N>`
A flag displaying whether the listener will delete the queue whenever a read occurs on the output file. Y will delete the queue, N will leave it in place.

`<H/M>`
A flag displaying the format of the output file. H is Human readable ASCII, and M is Machine readable binary.

5.9 Reading message queues

5.9.1 Human readable

Each listener creates a `/proc/ipd` file in order to provide an output mechanism for the messages it intercepts. The file has two available formats, the first one being human readable.

The format of received data in human readable form is:

TCP URGENT FORMAT:

```
Messages: #messages        Transport: TCP_URGENT
<msg#> - <secs>.<usecs> - <src ip>:<port> - <dst ip>:<port> - <msg><flag>
...
```

`<msg#>`
: Number of message as currently stored in the message queue.

`<secs>`
: Timestamp in seconds of when the packet was received.

`<usecs>`
: Timestamp in microseconds of when the packet was received.

`<src ip>`
: The ip address of the machine that transmitted the packet.

`<dst ip>`
: The ip address of the machine that the packet was intended for.

`<port>`
: The port number of the packet. The source port and destination port are attached to the respective ip addresses.

`<msg>`
: The actual message itself. Displayed as 2 characters.

`<flag>`
: A flag representing certain header flags of the packet.

IP DONT FRAGMENT FORMAT:

```
Messages: #messages        Transport: ID_DFRAG
<msg#> - <secs>.<usecs> - <src ip> - <dst ip> - <msg><flag>
...
```

`<msg#>`
: Number of message as currently stored in the message queue.

`<secs>`
: Timestamp in seconds of when the packet was received.

`<usecs>`
: Timestamp in microseconds of when the packet was received.

`<src ip>`
The ip address of the machine that transmitted the packet.

`<dst ip>`
The ip address of the machine that the packet was intended for.

`<msg>`
The actual message itself. Displayed as 2 characters.

`<flag>`
A flag representing certain header flags of the packet.

ARP PADDING FORMAT:

```
Messages: #messages        Transport: ARP
<msg#> - <secs>.<usecs> - <src ip> - <dst ip> - <msg>
...
```

`<msg#>`
Number of message as currently stored in the message queue.

`<secs>`
Timestamp in seconds of when the packet was received.

`<usecs>`
Timestamp in microseconds of when the packet was received.

`<src ip>`
The ip address of the machine that transmitted the packet.

`<dst ip>`
The ip address of the machine that the packet was intended for.

`<msg>`
The actual message itself. Displayed as a string of characters. Stored in a 60 character array.

5.9.2 Machine reaadable

The format of the machine readable queues is:

TCP URGENT FORMAT:

HEADER:

char
Transport Mechanism: **(2)**

int
Number of messages

MESSAGES:

int
Message Number

long
Timestamp (secs)

long
Timestamp (usecs)

4 bytes
Source IP address

4 bytes
Destination IP address

char
Message[0]

char
Message[1]

int
flag

IP DONT FRAGMENT FORMAT:

HEADER:

char
Transport Mechanism: **(1)**

int
Number of messages

MESSAGES:

int
Message Number

long
Timestamp (secs)

long
Timestamp (usecs)

4 bytes
Source IP address

2 bytes
Source Port

4 bytes
Destination IP address

2 bytes
Destination Port

char
Message[0]

char
Message[1]

int
flag

ARP PADDING FORMAT:

HEADER:

char
Transport Mechanism: **(0)**

int
Number of messages

MESSAGES:

int
Message Number

long
Timestamp (secs)

long
Timestamp (usecs)

4 bytes
Source IP address

4 bytes
Destination IP address

char[60]
Message

5.10 Clearing a message queue

Clearing a message queue is done by writing to the listener's output file. The filename is the one given when the listener was created, `/proc/ipd/<filename>`

6 C interface

A pair of C interfaces, one for each module, have been provided for easy integration of the Diffusion system into existing C and C++ programs. The interfaces take care of all `/proc` interaction and provide a more usable layer of abstraction.

6.1 Installation

The C interfaces can be found in the `lib/` directory of the Diffusion installation files. Each interface consists of a C source file and a header file. To compile the libraries as well as test programs for the libraries, change into the `lib/` directory and type `make`. This will create a pair of .o files that can be linked into your programs, as well as two executables which will test the corresponding systems when executed. Make sure the modules are installed before using the interfaces or running the tests.

6.2 Send interface

The C interface for the send module provides the ability to add entries to the send queues, clear the queues, and read the current contents of the queues. It also provides the ability to manipulate the ARP data that is sent. These features are implemented through a set of functions and data structures. To use these functions and structures in a program, include the provided header (`ipdsend_ctrl.h`) at the top of your source, and link in the file `ipdsend_ctrl.o`.

6.2.1 Data Structures

The following data structures are defined in the send interface header and are typedefed to remove the need for the `struct` keyword:

iptcpdata

```
typedef struct {
        char    message[2];
        unsigned char   flagip:1,
                        flagstcp:4,
                        unused:3;
} iptcpdata;
```

The `iptcpdata` structure provides a container for the data that can be sent over the IP or TCP channels in each entry. The `message` member contains the two bytes that will be put in the ID field of IP packets or the urgent pointer of TCP packets. The `flagip` and `flagstcp` members contain the data that will be put in the reserved flags of their respective packet headers.

iptcpentry

```
typedef struct {
        __u32 ipaddr;
        unsigned char mask;
        char repeat;
        char priority;
        iptcpdata data;
        char    useip:1,
                usetcp:1,
                unused:6;
} iptcpentry;
```

The `iptcpentry` structure provides a container for all the data put in the send queue for each IP/TCP entry. The `ipaddr` member contains the IP address, converted to a binary format. The `mask` member contains the mask for the IP address. The `repeat` and `priority` members are the repeat count and priority number given to the send module. The `data` member is an `iptcpdata` structure containing the data that will be sent for this entry. The `useip` and `usetcp` members are flags for whether this entry will be sent on the IP and TCP protocols respectively.

iptcplistentry

```
typedef struct iptcplistentry {
        iptcpentry* entry;
        struct iptcplistentry* next;
} iptcplistentry;
```

The `iptcplistentry` structure provides a singly linked list wrapper for the `iptcpentry` structure. This can be used to queue multiple entries, and is used in reading the queue. The `entry` member points to an `iptcpentry` structure, and the `next` member points to the next entry in the list. A value of NULL in the `next` pointer indicates the end of the list.

arpethdata

```
typedef struct {
        char length;
        char message[60];
} arpethdata;
```

The `arpethdata` structure provides a container for the actual data sent using the ARP and generic Ethernet protocols. The `length` member indicates how many bytes of data there are, and the `message` member is a buffer for the data. All data past `length` bytes of `message` is ignored.

ethentry

```
typedef struct {
        char macaddr[6];
        unsigned char mask;
        char repeat;
        char priority;
        __u16 protocol;
        arpethdata data;
} ethentry;
```

The `ethentry` structure provides a container for the data put in the queue for generic Ethernet entires. It contains the destination MAC address represented as 6 bytes of binary data in the `macaddr` member, and an IP style mask for the address in the `mask` member. The `repeat`, `priority`, and `protocol` members correspond to the repeat count, priority number, and protocol number given to the send module. The `data` member is an `arpethdata` structure containing the data to be sent for this entry.

ethlistentry

```
typedef struct ethlistentry {
        ethentry* entry;
        struct ethlistentry* next;
} ethlistentry;
```

The `ethlistentry` structure is a singly linked list structure identical to the `iptcplistentry` structure, except that it points to generic Ethernet entries rather than IP/TCP entries.

6.2.2 IP/TCP Functions

These functions manipulate the IP/TCP send queue.

ipd_create_iptcp_entry

```
iptcpentry* ipd_create_iptcp_entry(
        char* ipaddr,
        char mask
        char repeat,
        char priority,
        iptcpdata data,
        int useip,
        int usetcp);
```

This function creates an `iptcpentry` structure from the given data and returns a pointer to it. The `ipaddr` argument is interpreted as a dotted address (ie "127.0.0.1") and converted to the binary format. The `useip` and `usetcp` arguments are interpreted to be true if non-zero, and are put in the corresponding flags in the `iptcpentry` structure. The rest of the arguments are copied directly to the `iptcpentry` structure after checking for validity. If `ipd_create_iptcp_entry` encounters an error, it returns NULL.

ipd_enqueue_iptcp

```
int ipd_enqueue_iptcp(
        iptcpentry* entry);
```

This function adds a `iptcpentry` structure to the IP/TCP send queue. It returns zero on success, non-zero on failure.

ipd_enqueue_iptcp_list

```
int ipd_enqueue_iptcp_list(
        iptcplistentry* head);
```

This function adds a list of IP/TCP entries to the queue. The argument `head` should point to the first `iptcplistentry` structure in a list. The list should be terminated by a NULL pointer to the next element. `ipd_enqueue_iptcp_list` returns zero on success, non-zero on failure.

ipd_clear_iptcp

```
int ipd_clear_iptcp();
```

This function clears the IP/TCP send queue of all entries. It returns zero on success, non-zero on failure.

ipd_clear_ip

```
int ipd_clear_ip();
```

This function clears the IP/TCP send queue of IP entries only. It returns zero on success, non-zero on failure.

ipd_clear_tcp

```
int ipd_clear_tcp();
```

This function clears the IP/TCP send queue of TCP entries only. It returns zero on success, non-zero on failure.

ipd_iptcp_queue_length

```
int ipd_iptcp_queue_length();
```

This function returns the current number of entries in the IP/TCP send queue. It returns a negative number on failure, otherwise it returns the length of the queue.

ipd_iptcp_read_queue

```
iptcplistentry* ipd_iptcp_read_queue();
```

This function reads the current contents of the IP/TCP send queue and returns a pointer to the head of a linked list containing the queue. The function returns NULL on failure or if there are no entries in the queue.

ipd_iptcp_free_list

```
void ipd_iptcp_free_list(iptcplistentry* head);
```

This function frees the memory used by a list of entries and all of the entries it points to. Only use this if the structure was allocated with malloc (all structures created by the send interface are allocated with malloc).

6.2.3 Generic Ethernet Functions

These functions manipulate the generic Ethernet send queue.

ipd_create_eth_entry

```
ethentry* ipd_create_eth_entry(
        char* macaddr,
        char mask
        char repeat,
        char priority,
        __u16 protocol,
        arpethdata data);
```

This function creates an `ethentry` structure from the given data and returns a pointer to it. The `macaddr` argument is interpreted as a colon separated hexadecimal MAC address (ie ”12:34:56:78:9A:BC”) and converted to the binary format. The rest of the arguments are copied directly to the `ethentry` structure after checking for validity. If `ipd_create_eth_entry` encounters an error, it returns NULL.

ipd_enqueue_eth

```
int ipd_enqueue_eth(
        ethentry* entry);
```

This function adds a `ethentry` structure to the generic Ethernet send queue. It returns zero on success, non-zero on failure.

ipd_enqueue_eth_list

```
int ipd_enqueue_eth_list(
        ethlistentry* head);
```

This function adds a list of generic Ethernet entries to the queue. The argument `head` should point to the first `ethlistentry` structure in a list. The list should be terminated by a NULL pointer to the next element. `ipd_enqueue_eth_list` returns zero on success, non-zero on failure.

ipd_clear_eth

```
int ipd_clear_eth();
```

This function clears the generic Ethernet send queue of all entries. It returns zero on success, non-zero on failure.

ipd_eth_queue_length

```
int ipd_eth_queue_length();
```

This function returns the current number of entries in the generic Ethernet send queue. It returns a negative number on failure, otherwise it returns the length of the queue.

ipd_eth_read_queue

```
ethlistentry* ipd_eth_read_queue();
```

This function reads the current contents of the generic Ethernet send queue and returns a pointer to the head of a linked list containing the queue. The function returns NULL on failure or if there are no entries in the queue.

ipd_eth_free_list

```
void ipd_eth_free_list(ethlistentry* head);
```

This function frees the memory used by a list of entries and all of the entries it points to. Only use this if the structure was allocated with malloc (all structures created by the send interface are allocated with malloc).

6.2.4 ARP Functions

These functions manipulate the data that is broadcast in ARP packets.

ipd_set_arp_send

```
int ipd_set_arp_send(
        arpethdata data);
```

This function sets the broadcast ARP data to what is in the `data` argument. It returns zero on success, non-zero on failure.

ipd_clear_arp

```
int ipd_clear_arp();
```

This function clears the arp data, reseting it to zero length. It returns zero on success, non-zero on failure.

ipd_read_arp_send

```
arpethdata* ipd_read_arp_send();
```

This function reads the current arp data and returns a pointer to an `arpethdata` structure. The function returns NULL on failure.

6.3 Receive interface

The receive interface is a programmable interface into the receive module. The interface allows the creation and deletion of listeners, as well as control over the message queue itself. Messages picked up by the system can also be retrieved through the interface, and are returned in a message queue structure defined by the library.

6.3.1 Creating and deleting listeners

Listeners are managed with two functions provided from the library:

```
int ipd_add_receiver(const char * address,
                     int netmask,
                     message_type type,
                     char * filename,
                     int clear);
int ipd_delete_receiver(char * filename);
```

`ipd_add_receiver` adds a listener with the given definition parameters into the system. *address* is a string containing the ip address of the host or subnet that you wish to listen for in dotted notation. *netmask* is an integer value between 0 and 32 representing the netmask to apply to the given address. e.g. An address of 192.168.1.1 and a netmask of 24 creates a listener that accepts messages from the class C subnet of 192.168.1.*. *type* gives the transport mechanism that the listener will listen on. The available values are `IP_DFRAG, ARP, and TCP_URGENT`. *filename* is the filename you wish the results to be reported to in `/proc/ipd/`. If the filename is not unique to the system the call will fail, and return in error. *clear* is a flag that accepts either 1 or 0. It controls whether or not the message queue is deleted every time a set of messages is retrieved from the system. If set to 1 the queue will clear, if 0 it will not be affected.

`ipd_delete_receiver` deletes a listener from the system. *filename*, its only parameter, is the name of the file in `/proc/ipd/` that it outputs the messages to. This should be the same filename that was used in `ipd_add_receiver`

Both functions return -1 on error and 0 on success

6.3.2 Control the queues

```
int ipd_delete_queue(char * filename);
```

`ipd_delete_queue` clears the queue of the specified listener. *filename* is the name of the file in `/proc/ipd/`.

```
void ipd_free_queue(ipd_receiver_t * q);
```

`ipd_free_queue` deletes an ipd receiver structure after it has been retrieved and used. The difference between this function and the `ipd_delete_queue` function is that delete clears the queue in the module, while free deletes the structure holding the retrieved messages.

6.3.3 Retrieving messages

```
ipd_receiver_t * retrieve_messages(char * filename);
```

`retrieve_messages` reads all the messages from a message queue and returns them in the form of a predefined structure. *filename* is the name of the file in `/proc/ipd`. The layout of `ipd_receiver_t` is included below. If the listener was created with the *clear* flag set to 1, then the system queue will be cleared after the messages have been read in.

```
typedef struct ipd_receiver_t {
        int num_msgs;
        char * filename;

        message_type type;
        message_queue_t *msg_queue;
        message_queue_t *msg_queue_tail;
} ipd_receiver_t;
```

The receiver structure contains the message queue as well as information about the listener. `num_msgs` provides the number of messages contained in the queue. `filename` is the name of the file in `/proc/ipd/` that the messages were read from. `type` is the transport mechanism that the listener is using. Its values will either be `ARP, IP_DFRAG,` or `TCP_UNIQUE`. The message queue itself is kept in a doubly linked list with pointers to the front and back given by `msg_queue` and `msg_queue_tail` respectively.

The message queue itself is just a doubly linked list where each element contains a pointer to a message structure that represents one message that was received by the listener.

```
typedef struct message_queue_t {
        message_t msg;
        struct message_queue_t * next;
        struct message_queue_t * prev;
} message_queue_t;

typedef union message_t {
        tcp_message_t * tcp_msg;
        ip_message_t * ip_msg;
        arp_message_t * arp_msg;
} message_t;
```

The message structures returned by the interface have the following form:

```
typedef struct tcp_message_t {
        __u32 saddr;
        __u32 daddr;
        int srcport;
        int dstport;

        int flag;
        char message[2];
        struct timeval timestamp;
} tcp_message_t;


typedef struct ip_message_t {
        __u32 saddr;
        __u32 daddr;

        int flag;
        char message[2];
        struct timeval timestamp;
} ip_message_t;

typedef struct arp_message_t {
        __u32 saddr;
        __u32 daddr;

        char message[60];
        struct timeval timestamp;
} arp_message_t;
```

`saddr` is the ip address of the sender and `daddr` is the ip address of the destination host. `timestamp` is the time when the message was received, in *sec* and *usec*. `message` is the actual message that was received by the listener. `flag` tells whether packet header flags were set or not. `srcport` is the tcp port used by the sending host and `dstport` is the tcp port used by the receiving host

7 Applications

This section describes the tools built on top of Diffusion.

7.1 SpyTalk

SpyTalk lets you communicate unidirectionally and bidirectionally using Diffusion. You can use it non-interactively in a way similar to netcat (nc), and interactively in a way similar to talk.

7.1.1 Installation

To install SpyTalk you will need to have root access. The installation process has been automated for you. There are only two steps you will need to perform. First, go into the directory where SpyTalk has been unpacked (probably the SpyTalk subdirectory of where Diffusion was unpacked), and type `make`. This will connect to CPAN, the Comprehensive Perl Archive Network, to make sure that you have the modules needed by SpyTalk. You will need Internet access to complete this step. If you have never used the CPAN module before, it will ask you if you want to configure it. In most cases you should be able to type "no" and have it autoconfigure. It will then check the versions of the modules needed by SpyTalk and upgrade them if necessary.

The second step is to type `make install`. This is optional, and will simply put SpyTalk in the `/usr/local/bin` directory and install a man page.

7.1.2 Synopsis

- **SpyTalk -t|--talk [-q|--quiet] [-s|--switch|--tcp] [-f|--file** *file* **[-c|--continuous|--noautorate] [-r|--rate** *rate*`]] [-p|--`**priority** *priority*`]` *ipaddr*
- **SpyTalk -l|--listen [-q|--quiet] [-s|--switch|--tcp] [-f|--file** *file*`] [-n|--`**timeout** *timeout*`]` `[`*ipaddr*`[`*/mask*`]]`
- **SpyTalk [-i|--interactive] [-s|--switch|--tcp] [-q|--quiet]**

7.1.3 Description

SpyTalk is a messaging front end to the Diffusion kernel modules. SpyTalk supports three modes of operation: talk, listen, and interactive. If no mode is specified, SpyTalk defaults to the fullscreen interactive mode. In talk mode, SpyTalk normally sends all input from STDIN to the computer specified by *ipaddr*, until you type ^D. In listen mode, SpyTalk normally listens for all input from any source and writes everything it receives to STDOUT until it receives a ^D on STDIN. However, if *ipaddr* is specified, SpyTalk will listen only to that IP address, and if *mask* is specified, *ipaddr* will be treated as a network with the given mask. Interactive mode is the most capable mode of SpyTalk. For a description of this mode and a list of commands, see below.

7.1.4 Options

-i, --interactive
Put SpyTalk into interactive mode. This is the default. not necessary.

-l, --listen
Put SpyTalk into listen mode.

-t, --talk
Put SpyTalk into talk mode.

Talk mode options:

-c, --continuous, --noautorate
Do not automatically adapt the send rate to avoid large send buffers when sending files. Queue everything as quickly as possible.

-f, --file *file*
Send the specified *file* over the channel and then quit, ignoring all keyboard input and ^D's.

-p, --priority *priority*
Uses the specified *priority* in sending the data rather than the default of 0. Range is −128 to 127.

-q, --quiet
Quiet mode. Suppresses all messages from SpyTalk.

-r, --rate *rate*
Queue *rate* bytes per second when sending files. Use this option when you know how much traffic will already be present on the network. Note that each byte of the file is sent five times, so it is recommended that you set the rate to one fifth the rate at which you expect to be sending packets.

-s, --switch, --tcp
Uses TCP rather than IP to send messages.

Listen mode options:

-f, --file *file*
Writes all data received to the specified *file* rather than to STDOUT.

-n, --timeout *timeout*
Ignores all keyboard input and instead waits *timeout* seconds and then quits.

-q, --quiet
Quiet mode. Suppresses all messages from SpyTalk.

-s, --switch, --tcp
Listens for messages on TCP rather than IP.

Interactive mode options

-q, --quiet
Quiet mode. Suppresses introduction and all messages from SpyTalk.

-s, --switch, --tcp
Switches the protocols, using TCP for messages and IP for the load.

7.1.5 Interactive Mode

In interactive mode, you are presented with a screen with two panels, as you can see in Figure 2. The top panel contains messages received from other computers, and messages from SpyTalk. The bottom panel contains everything you type. In addition to transmitting messages, SpyTalk also sends the computer's current load in interactive mode. The current load for the local and remote computers are displayed above their respective message panels.

Interactive mode is controlled by various commands. The commands are case insensitive. The following commands are currently understood by SpyTalk:

talk *host*
Open a connection to the computer with the specified IP address or hostname. This opens a connection for sending and receiving messages.

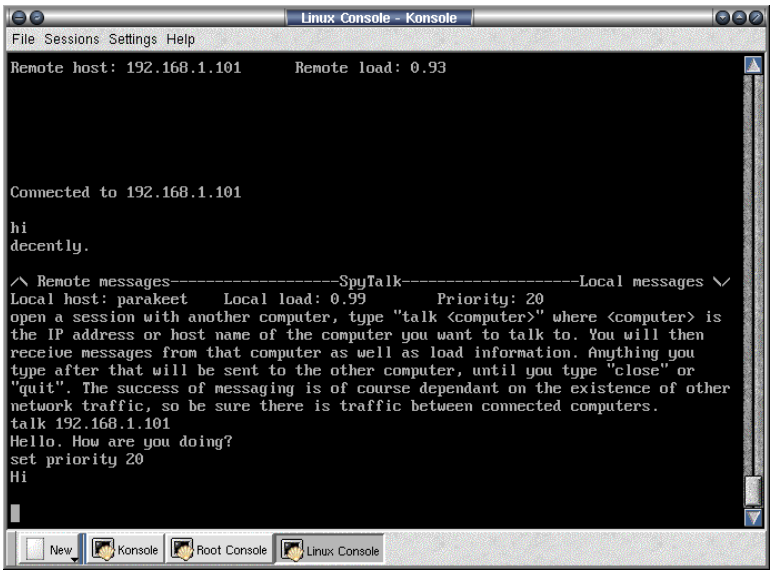

Figure 2: Example of Spytalk running interactively.

set priority *number*
: Set the sending priority for future messages to the specified number. Range is −128 to 127.

close
: Close the currently open connection

quit
: Close any open connection and quit SpyTalk.

7.1.6 Notes

Because of the nature of Diffusion, SpyTalk does not actually create any network traffic. If you expect to be sending and/or receiving messages but they are not going through, check to make sure there is other network traffic between the source and destination.

7.2 LoadBanner

In LoadBanner, machines advertise their current load to the network using Diffusion. Clients can then read the load using Diffusion. In this way, load information is communicated without consuming any network bandwidth. If the Diffusion mechanism being used is ARP, then a server makes its load known to all the clients on the LAN without consuming any bandwidth.

7.2.1 ldavgd

Ldavgd is a small program that continuously reads the load average from a machine and then queues it for transport over the send module.

```
ldavgd [-i interval] [-t transport] [-q] [-l level] <ip address>
```

`<ip address>`
: The ip address of the host that the measurements will be sent to.

`-i interval`
The time interval that the program will wait between taking load measurements, and queuing them for transmission.

`-t transport`
Specifies the transport mechanism. Values can be `ARP`, `IP_DFRAG`, or `TCP_URGENT`. Default is `TCP_URGENT`.

`-q`
Quiet mode. When supplied queuing notifications are not outputted to the screen.

`-l level`
The load average which you wish to transmit. Either `1`, `5`, or `15`.

7.2.2 ldavgc

Ldavgc is the client program for LoadBanner. It displays the latest average from all the reporting computers in a color terminal window.

```
ldavgc [-i interval] [-t transport]
```

`-i interval`
The time interval that the program will wait between retrieving measurements from the receive queue.

`-t transport`
The transport mechanism that the program will look for measurements on. Values can be `ARP`, `IP_DFRAG`, or `TCP_URGENT`. The default is `TCP_URGENT`.

7.2.3 ldavgc interface

The LoadBanner client allows the user to monitor many different hosts simultaneously. Multiple hosts can be monitored simultaneously by the program using any of the 3 supported protocols. Each of the host definitions can include a subnet range, to monitor multiple hosts in the same network block, as well as a protocol identifier to specify which communication channel the specific host will be transmitting on.

When `ldavgc` is executed it displays a blank screen and waits for the user to input a host definition. Hosts are added with CTRL-A. A window appears and requires the user to supply an ip address and netmask and potentially a protocol. Any host in that range that is sending data will then appear on the screen. Hosts are deleted with CTRL-D, a list of active listeners is displayed and the user chooses the number of the listener to delete. Color codes are used to express the value of the load average as well as the age of the measurement. If the screen dimensions are too small to include all the hosts being received, the user can cycle through the list with CTRL-N and CTRL-P. The client can be exited from at any time by pressing q.

8 Performance

The bandwidth and latency at which Diffusion can send data is dependent on the communication protocols that it piggybacks on. It is difficult to determine at which point the Diffusion implementation itself constrains these numbers. In practice, it appears that it never does.

We have carefully measured the overheads involved with using Diffusion: the degree to which it impacts the performance of the system as a whole and the performance of the network stack. For the first, we timed

Description	Performance Impact	p-value of difference
Baseline	0%	n.a.
Baseline + modules	-0.9%	0.02
Baseline + receive module + one listener	-1.1%	< 0.001
Baseline + send module + 500 send entries (one on)	+4.1%	< 0.001

Figure 3: Impact of Diffusion on system performance.

Description	Transfer Rate Impact	p-value of difference
Baseline	0%	n.a.
Baseline + modules	+0.4%	not significant
Baseline + receive module + one listener	+1.9%	not significant
Baseline + send module + 100 send entries (one on)	-12.0%	test run once
Baseline + send module + 500 send entries (one on)	-25.0%	< 0.001

Figure 4: Impact of Diffusion on network performance.

the repeated compilation of the Linux kernel under various conditions. For the second, we measured the bandwidth between two computers using Netperf (www.netperf.org) under various conditions. All tests were run while ping flooding between two computers using the modules, to simulate an environment with heavy network traffic.

Our measurements were done using Red Hat Linux 7.2 with a Linux 2.4.7 kernel. The test systems have two Pentium III processors each running at 1 GHz, 1 GB of system RAM, and was using an Intel EtherExpress 10/100 network card. The two systems were kept on an isolated network.

8.1 System overhead

The effect that Diffusion has on system performance is dependent on what the modules are doing. Because of this, we have tested the performance of our system in four different configurations for comparison: without the modules, with both modules but no listeners and an empty queue, with just the receive module and a listener that is receiving data, and with just the send module and a 500 entry queue. Each configuration was tested nine times to get a more accurate sample, and the testing rotated among the configurations at each iteration so there would not be any chance of time biased data (computer slowing down with time or whatnot). Our metric is the extent to which a kernel compilation was slowed down.

Figure 3 illustrates the results of our study. The modules themselves decrease performance by less than 1%. As more listeners or send queue entries are added, performance is impacted. However, at some point, performance slowly increases again. With 500 outstanding send requests, there is a 4.1% performance *increase.* over the baseline case. We speculate that this strange behavior might be explained by the significant decrease in network performance in this scenario. If the system cannot send out packets as quickly, it may have more processor time free for computation.

8.2 Network overhead

The effect of Diffusion on network performance was tested in the same way as system performance. In each of the configurations, Netperf was asked to measure the available network bandwidth and the transit rate that the two machines was able to sustain. Figure 4 illustrates this study. Essentially, loading the modules or running a few listeners has negligible impact on network performance. On the other hand, having a large

number of data items queued for sending does impact performance significantly. This is almost certainly due to the linear search algorithm that the implementation uses.

9 Conclusion

We have described in detail the interface and implementation of Diffusion within the Linux kernel and tools built on top of it. We have demonstrated that it is feasible to implement this concept within a modern operating system kernel and that it can have minimal overhead on the system provided that the number of outstanding communications is kept reasonably low. Even that caveat is an artifact of the fact that our implementation currently uses linear search to do longest prefix matching, which can be replaced with known logarithmic time and expected constant time approaches. We also showed how applications can use the communication channel provided by Diffusion to do surreptitious communication and to communicate resource information while consuming no additional network resource beyond those used for packet transfers that otherwise occur. We are currently investigating the integration of Diffusion-based communication into the RPS system.

References

[1] DINDA, P. A. Exploiting packet header redundancy for zero cost dissemination of dynamic resource information. In *Proceedings of the 6th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computer (LCR 2002)* (May 2002). To Appear.

[2] DINDA, P. A., AND O'HALLARON, D. R. An extensible toolkit for resource prediction in distributed systems. Tech. Rep. CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, July 1999.

[3] LOWEKAMP, B., MILLER, N., SUTHERLAND, D., GROSS, T., STEENKISTE, P., AND SUBHLOK, J. A resource monitoring system for network-aware applications. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (July 1998), IEEE, pp. 189–196.

[4] TIERNEY, B., AYDT, R., GUNTER, D., SMITH, W., SWANY, M., TAYLOR, V., AND WOLSKI, R. A grid monitoring architecture. Tech. Rep. GWD-GP-16-2, Global Grid Forum Performance Working Group, March 2000.

[5] WOLSKI, R. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings of the 6th High-Performance Distributed Computing Conference (HPDC97)* (August 1997), pp. 316–325. extended version available as UCSD Technical Report TR-CS96-494.